Spring 2018

IMAGE TO LATEX VIA NEURAL NETWORKS

Avinash More
San Jose State University

IMAGE TO LATEX VIA NEURAL NETWORKS

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment of

the Requirements of the Degree

Master of Science

By

Avinash More

May 2018

SAN JOSÉ STATE UNIVERSITY

The Undersigned Thesis Committee Approves the Thesis Titled

IMAGE TO LATEX VIA NEURAL NETWORKS

By

Avinash More

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Chris Pollett, Department of Computer Science

Dr. Robert Chun, Department of Computer Science

Mr. Varun Soundararajan, Google

ACKNOWLEDGEMENT

I would like to express my gratitude to my project advisor Dr. Chris Pollett for his support and the guidance. I would not have been able to complete this project without Dr. Pollett's valuable suggestions. I would also like to thank my committee members Dr. Robert Chun and Mr. Varun Soundararajan for their suggestions and time.

I am also thankful to my friends and family for all the moral support and constant encouragement.

ABSTRACT

Many research papers in mathematics, computer science, and physics are written in LaTeX. Technical papers and articles in these areas often involve mathematical equations. Writing such equations in LaTeX takes longer than handwriting the same equations on paper. In this report, we want to show that the time-consuming process of typesetting LaTeX equations from images of these equations can be automated and optimized. Neural networks are good at solving related problems such as handwritten digit recognition, so we adapted these well-studied approaches to the LaTeX problem. Neural network model training involves large amounts of good quality data. So, for our project, we propose a convolutional neural network architecture to recognize LaTeX equations along with a way to generate labeled datasets of mathematical equation images and their corresponding LaTeX expressions. Our neural network model predicts from mathematical equations involving numbers, letters, mathematical symbols, and matrix images, the corresponding LaTeX for these equations. We have achieved an accuracy of more than 90% in predicting LaTeX for these complex equations involving up to 35 characters.

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

Many research papers in mathematics, computer science, and physics are written in LaTeX format. While writing technical papers or articles, there are some scenarios where the text to be written is a mathematical equation. Writing a mathematical equation in LaTeX format takes a lot more time compared to writing the same equation on paper. The time-consuming approach of converting the equation written on paper to LaTeX format can be automated and optimized. In this project, I am exploring an approach to automate this process.

Getting a LaTeX representation for a mathematical equation is a harder problem to the optical character recognition (OCR) problems which have been attempted in the past. There are multiple reasons why this problem is harder.

1. **Size**

   The size of a character varies depending on the position of the literal in the mathematical equation.

   For example:

   a. Size of a character in the exponent part is smaller than the size of the character in the base part. Example: a^a

2. **Fonts**

   In a mathematical equation, the font of the variable and font of the functions can be different.

   Examples:

   a. Log functions:

$$\log xy = \log x + \log y$$

Figure 1: Log function

   b. Limits:

$$\frac{df}{dt} = \lim_{h \to 0} \frac{f(t+h) - f(t)}{h}$$

Figure 2: Limits

3. **Multiline equations**

   A single equation can have multiple lines.

   Examples:

$$n = \cfrac{1}{\frac{2n-1}{n} + \cfrac{1}{\frac{2n-3}{n} + \ddots + \cfrac{1}{\frac{2n-n}{n}}}}$$

Figure 3: Multiline mathematical equation

4. **Equation reading order**

   In a usual OCR problem, we read line by line. We read each line from left to right. After the end of the line, we move to the next line and read that line from left to right again. So, we have this predefined setup in place where we read from top to bottom for lines and from left to right for characters in a line. In mathematical equations, this is not always the case. Let us take a look at an example of an equation containing a matrix.

$$\begin{bmatrix} 1 & 2 \\ b & 3 \end{bmatrix} = \begin{bmatrix} a & 5 \\ x & q \end{bmatrix} + \begin{bmatrix} w & m \\ 5 & d \end{bmatrix}$$

Figure 4: Equations containing matrix

LaTeX corresponding the above equation is “$ \begin{bmatrix}1 & 2\\ b & 3\end{bmatrix} = \begin{bmatrix}a & 5\\ x & q\end{bmatrix} + \begin{bmatrix}w & m\\ 5 & d\end{bmatrix}$ ”. If we analyze this expression we can find few things. If this was

a normal OCR implementation, then it would have considered this equation to be of two lines. The first line would have been the first row all three matrices. The second line would have consisted "=" and "+". The third line would have consisted of the second row of all the matrices. Contrary to this way, this mathematical equation is read differently as we can see from the LaTeX expression. We read the whole matrix on the left-hand side of assignment operator before moving to the assignment operator. While reading the matrix we would read the starting matrix symbol which lasts for two vertical lines, then we read the first row of the first matrix from left to right, then we move to next row of the matrix and then we read the closing matrix bracket. After reading the matrix in this fashion we read the assignment operator and the other two matrices in the same fashion.

Reading text from the images of books is not a new problem. Problems like this have been attempted since the early 1900s. In the year 1914, a machine was developed by a Russian researcher Emanuel Goldberg to read the characters and convert those to telegraph code [1]. In 1974 researchers Ray Kurzweil started working on a product which should be able to recognize character written in any font also called Omni-font OCR problem. In 1978, the first of commercial version of this OCR computer software was introduced. As of the present, with the popularity of the internet, many online cloud-based services are available for OCR. Further with the use of smartphones, OCR applications have become commonplace. Different OCR applications try to solve specific OCR problems. Common examples include reading financial documents such as Cheque, automatic character recognition from the vehicle number plate, reading text from the images of books (Google books), etc.

Reading mathematical expression from an image has been attempted as early as 2000s (Belaid and Haton 1984). In this paper, they use a syntactic parser to interpret 2-D mathematical

formulas. In this approach, they parse using a localizing a principle operator contained in the formula and from there they partition it into subsequences where each subsequence is analyzed similarly [2].

Winkler et al. 1995 [3] proposed a method to find symbols in handwritten mathematical expression images based on Hidden Markov Model (HMMs). In this method, once a symbol was identified actual classification would be done by finding the most probable symbol sequence based on HMM. Consider an example where a period symbol can be a decimal point as well as it can be a multiplication operator. Once a period symbol has been recognized, it was classified as period symbol as a multiplication symbol, or it was a period symbol as it is in case of decimal point using HMM.

Unlike a problem of getting a mathematical expression from an image, an image to markup generation problem is a relatively new problem. For example, in the year 2016 this problem statement was also posted on the OpenAI's requests to research section. One of the approaches used to solve this problem was using a concept called coarse-to-fine attention by Harvard researchers. In this project, a dataset of images used was the dataset formed by extracting the mathematical equations from the different technical publications [4].

A similar problem to our problem is that of converting an image to a corresponding HTML. Deng, Yuntian et al, 2016 explored an approach to solve this problem with a model which does not need to have any knowledge of the underlying markup language. This model is an extension of attention based encoder-decoder model.

All the solutions to the above application specific OCR problems were based on the technology trends of the time period in which they were written. In recent years, because of advancement in the hardware technologies, the training of neural networks has become more

feasible and easier. In this project, we are trying to solve the LaTeX Transcription Problem(LTP) with the neural networks. We use convolutional neural networks (CNN)to solve LTP.

This report is organized into six different sections. The first section deals with the introduction and already existing publications related to LTP. In the second section, we discuss background needed to implement and understand the proposed solution. The third section explores the data generation approach we have used. The implementation details involving all the four deliverables are discussed in the fourth section. Experiments and results are the part of section five. Section six concludes the report.

BACKGROUND

In this section of the report, we will discuss the back ground required to implement the solution we are proposing. We will explore neural networks, convolutional neural networks, and deep learning library Tensorflow.

Neural networks are loosely modeled after the human brain. In the human brain, we have millions of neurons which are connected to each other. So, it is basically a network of neurons. These neurons receive signals from inputs or neuron from earlier layers. Those signals are passed along to appropriate neurons based on the type of the signal they carry.

Consider the artificial neural network in Figure 5. As shown in the image, this neural network has five layers. The first layer is called an *input* layer. The last layer is called as the *output* layer. All the layers excluding the first and the last layer are called as *hidden* layers.

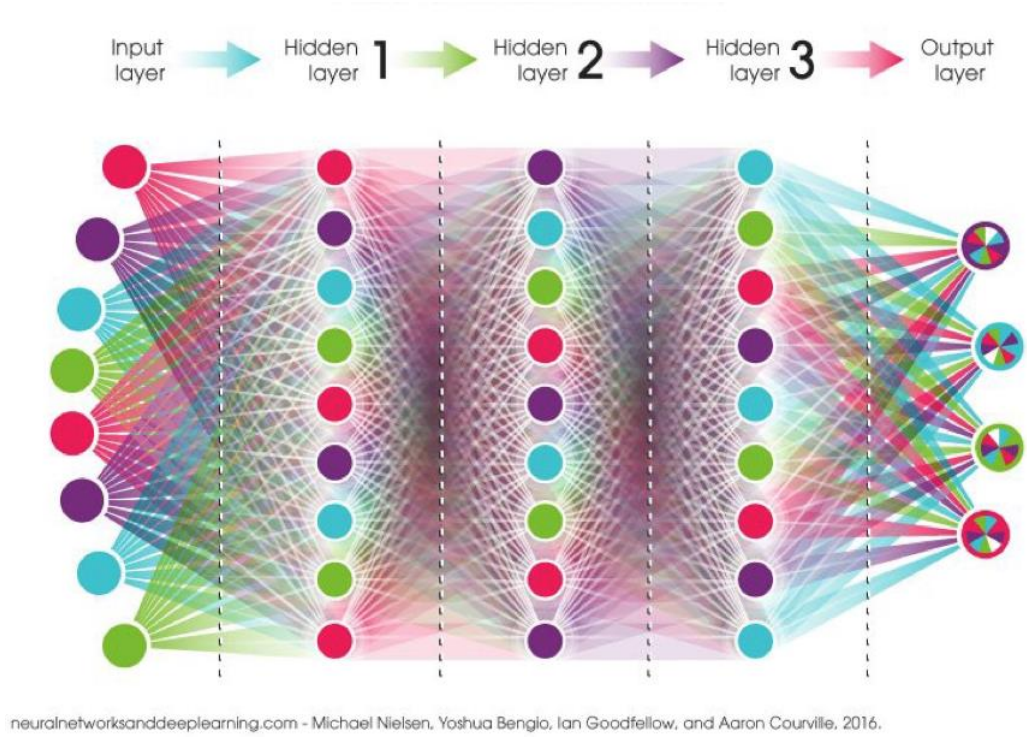

Figure 5: Deep Neural Network

We can see from the image that all the nodes are connected to all the nodes in the next layer. All the edge connection from nodes in one layer to nodes in the next layer are have weights. The decision of whether the neuron would fire or not is taken as follows.

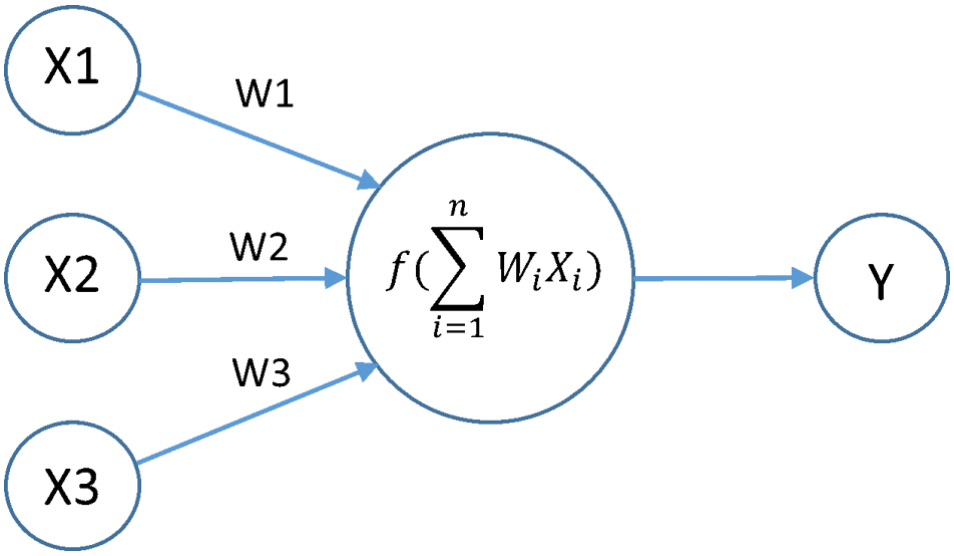

Figure 6: Single Neural Calculation

How the output at each node is computed as shown in Figure 6. The weighted sum is taken and a non-linear function is applied to that sum to decide whether the neuron would fire or not. The typical non-linear function choices are a sigmoid function, tanh function, rectified linear unit(RELU) function etc.

Now that we have explored neural networks a little bit, let us jump directly into the convolutional neural network. The convolution operation in deep learning is inspired from the convolution operation in digital signal processing. In digital signal processing, a convolution of two signals is used to get the third digital signal. Convolution can be considered as an algorithmic procedure of mixing information from multiple sources to get the desired output. Similarly, in deep learning convolution operations can be used to mix information from multiple sources to get desired results. Convolution is the fundamental operation on which convolutional

neural networks are based. CNNs are the most popular neural networks for computer vision tasks.

In a CNN, we use a small size window to focus on only a certain part of the image at a time. We use the same window to slide across the whole image to identify various features. This small window is called as feature map or a kernel or a filter. Usually, convolution is used with a pooling operation to extract the important features from the convoluted output. Typical CNN implementation looks like as shown in Figure 7.

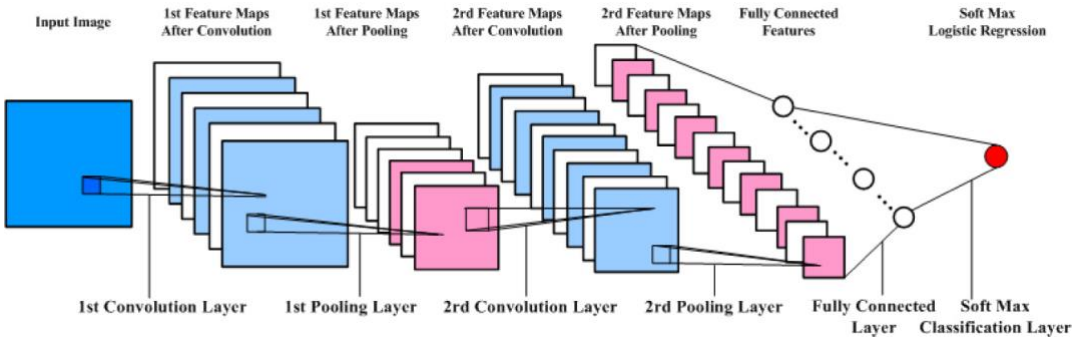

Figure 7: Convolutional neural network

As shown in Figure 7, we try to identify multiple features at each layer of CNN. Let us see how convolution operation is performed.

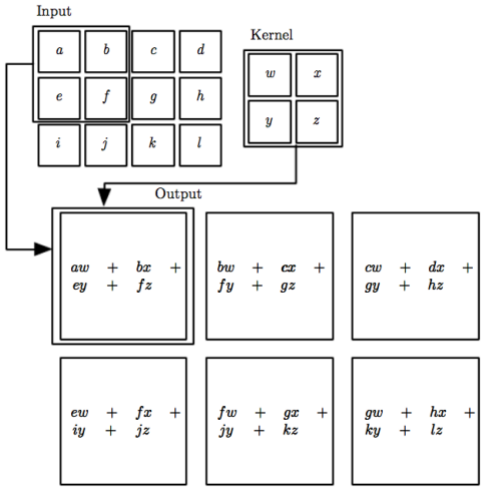

Figure 8: The convolution operation

To perform convolution operation, we take kernel and map it on the input and take dot product imposed image and the kernel. Then we slide the kernel across the whole image to get the result corresponding to the next superimposed part of the image. This operation is showsn in the Figure 8.

Now that we have understand how a convolution operation is performed, let us see how the pooling operation is performed. In our example, we are performing max pooling with a kernel of size 2×2. We take the maximum element from the first 2×2 window of the image. We move the kernel to right by two positions and take max value again. This procedure is repeated multiple times till we cover the entire image. As shown in Figure 9, the red colored box on the left side of the image is max-pooled to get the value of 12 which is in the red box on the right side of the image.

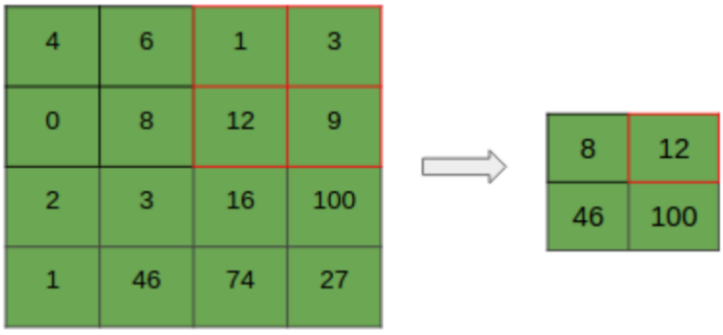

Figure 9: Max pooling operation

Tensorflow

After deciding on the choice of neural network it seemed essential to explore a suitable deep learning library. We explored a deep learning library by Google called TensorFlow. The following are some reasons to use an already existing popular library:

- **Code reuse:** It is better not to reinvent the wheel.
- **Less error-prone:** As the libraries are used and tested by multiple users it is usually less error-prone.
- **Level of abstraction:** Using a library provides a level of abstraction because of which programmer does not have to worry about minor details of the code.
- **Efficiency:** Libraries are written by the expert programmers. They are tested for performance. Therefore, libraries are efficient.
- **Online community support:** Because popular libraries are used by a large community of programmers, the community provides excellent online support.

Python programming language has libraries such as NumPy and SciPy to make it more efficient for numerical and scientific computing. NumPy has its subroutines compiled in a more efficient the low-level programming language. Therefore, using NumPy subroutines instead of Python looping constructs with lists makes Python programs more efficient. Transferring data from one programming environment to other is an expensive operation. This overhead gets even more when we are transferring data to graphics processing unit (GPU) or when we are transferring the data to a distributed environment where there is a high cost associated with the data transfer. To handle this overhead TensorFlow has an interesting approach. Instead of frequently switching between different language environments, TensorFlow lets a user define the complete computation. This complete computation is represented by computation graph of

interacting operations. Once the complete graph is defined, TensorFlow executes the complete computation graph by establishing a session.

There might be a scenario when we do not know the input at the time of deciding the computation graph. To handle such scenarios TensorFlow has a concept called placeholders. Using placeholders, input can be fed to the computation graph.

TensorFlow also has various methods for doing common operations related to multilayer perceptron such as follow:

1. **Calculating the error or the cost**:

   *cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))*

2. **Applying various matrix operations such as multiplication, addition, inverse, etc**.:

   y = tf.matmul(x,W) + b

3. **Applying different activation functions:**

   h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)

4. **Deciding and applying the backpropagation algorithms:**

   train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

5. **Defining a neural network:**

   ```
   def conv2d(x, W):
     return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')


   def max_pool_2x2(x):
     return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                           strides=[1, 2, 2, 1], padding='SAME')
   ```

TensorFlow has provides a way to save and restore the model.

The following code can be used to save the model:

```
with tf.Session() as sess:
    sess.run(init_op)
    # Do some work with the model.
    inc_v1.op.run()
    dec_v2.op.run()
    # Save the variables to disk.
    save_path = saver.save(sess, "/tmp/model.ckpt")
    print("Model saved in file: %s" % save_path)
```

The following code can be used to restore the model:

```
with tf.Session() as sess:
    # Restore variables from disk.
    saver.restore(sess, "/tmp/model.ckpt")
     print("Model restored.")
```

We explored these methods of TensorFlow with the classic MNIST dataset.

DATA GENERATION

The importance of finding the correct and appropriate dataset which will be closer in representation to the real-world data is not often recognized. But, there have been many instances where the same algorithm with a better set of data has given much better results [6]. Therefore, it was important to find appropriate data for training the model. After exploring various standard machine learning data libraries such as UCI machine learning repository, kaggle.com, deeplearning.net, etc., it was realized that coming up with a way to generate data and labels would be a better way moving forward. Therefore, we explored a couple of approaches to generate data.

Two approaches were explored prominently. The first approach was creating a portable network graphics (PNG) image file and the second approach was for generating portable document format (PDF) file using postscript. The approach of generating PDF uses PNG as base file and embeds it with more wrappers and creates an equivalent PDF file. Therefore, using PNG file seemed more appropriate way. So, we started exploring various ways to create a PNG image of mathematical equations.

The PNG image file can be generated with a code in different ways but for our dataset generation, there are multiple conditions which need to be satisfied. One of the most important condition while generating a data set for training is that it should have image representation as well as the label for the same. The label for this problem is a LaTeX representation of the image. Having an exact representation of the LaTeX for all the images in the dataset is a critical requirement for our problem-solving approach.

While exploring possible libraries for generating data library modules Matlab seemed suitable. As Matlab is a proprietary, its corresponding library in Python programming language called Matplotlib was chosen. Using Matplotlib images can be created with an equation in LaTeX form as the text.

Following code snippet shows the way to generate images:

```
def render_latex(formula, fontsize=10, dpi=300, format_='svg'):
    fig = plt.figure(figsize=(0.3, 0.25))
    fig.text(0.05, 0.35, u'${}$'.format(formula), fontsize=fontsize)
    buffer_ = StringIO()
    fig.savefig(buffer_, dpi=dpi, transparent=True, format=format_, pad_inches=0.0)
    plt.close(fig)
    return buffer_.getvalue()


image_bytes = render_latex(expression, fontsize=5, dpi=200, format_='png')
image_name = './data/' + 'name.png'
with open(image_name, 'wb') as image_file:
    image_file.write(image_bytes)
```

While using matplotlib for creating images and corresponding LaTeX labels works well when the equations do not have a complex mathematical representation such as a matrix. While rendering these non-complex mathematical equations we do not need to have TeX installed, since matplotlib has its own TeX expression parser along with layout engine, and fonts. The layout engine used in matplotlib TeX is a fairly direct adaptation of the layout algorithms in

Donald Knuth's TeX, so it has a good quality while rendering the LaTeX in the image format. For complex mathematical equations rendering, we have to have TeX installed on the machine for python to use it. To use installed TeX we have to specify it in configurations setting part of the code. The following two lines can be used to set this parameter.

```
from matplotlib import rcParams
rcParams['text.usetex'] = True
```

After setting these parameters it is important to use 'amsmath' package which provides the support for representations such as a matrix. The following line of code can be used to refer to 'amsmath' package.

```
rcParams['text.latex.preamble'] = r'\usepackage{amsmath}'
```

Another important condition which should be satisfied while using these lines of code is that path of LaTeX installable should be set correctly to make these lines work. The following line will append already existing environment path variable with LaTeX path

```
os.environ["PATH"] += os.pathsep + '/Library/TeX/texbin'
```

Once, all these setting are done any text element can use math text. You should use raw strings (precede the quotes with an 'r'), and surround the math text with dollar signs ($), as in TeX.

IMPLEMENTATION

Image to LaTeX is a complex problem to solve. This problem has multiple challenges. Those challenges we have already discussed in the previous sections. Because of these challenges, it was important to divide the problem into small chunks and conquer it individually.

For our solution, we divided the problem solution into four deliverables. Each deliverable is built upon the previous deliverable. Following are the five deliverable which we worked on during the implementation phase.

1. The first deliverable: Predicting the first character
2. The second deliverable: Predicting the LaTeX for simple equations
3. The third deliverable: Predicting the LaTeX for complex mathematical equations
4. The fourth deliverable: Predicting the LaTeX for matrix operations [2 X 2]

In the next few pages, we will discuss each of these deliverables in details. We will also discuss the challenges which we faced and approaches which helped us to tackle these challenges.

Predict the first character

Prediction of the first character in a long mathematical equation seemed like the most basic step while starting with the implementation of a solution to an image to LaTeX problem. This deliverable can be considered as a foundation of the solution which we have proposed in this project.

While implementing this deliverable, we started with images of equations consisting of five to ten characters. We did multiple versions of this simple deliverable. Initially, we generated equations starting with only four possible characters as the first characters. Those four characters were a summation symbol ($\sum$), an integration symbol ($\int$), a square root symbol ($\sqrt{}$), and a fraction symbol. For the first three choices, it can be easily understood that the first symbol is as expected or as it appears in the image if it is assumed that reader is reading the image from left to right and top to bottom. In case of a fraction, the numerator appears as the first symbol when we are reading it, but in LaTeX code it is a different scenario. Let us take a look at why for fractions it is not so straightforward.

Consider a fraction $\frac{a}{b}$. In this example the first symbol for a reader is a letter 'a', the second symbol is a horizontal bar and the third symbol is a letter 'b'. When we write corresponding equation in LaTeX, its representation is as *\frac{a}{b}*. In the LaTeX code for same fraction, the first character is \frac. Therefore, even for a simplest example this is not as straightforward at it seems.

The first decision we made while working on this deliverable was on the representation of the labels. It is a very common and a popular technique to use one hot vectors encoding to

represent labels in classification problems. While using a one hot vector encoding technique, we set all values in the vector to zero except for the label of the current class.

The following table shows the image and a label encoding with a few examples.

Table 1: Image and the Label Representation

Image	Label corresponding the first character
$\sqrt{c}$	[1,0,0,0]
$\int_q^k a^2 da$	[0,1,0,0]
$\sum_{j=1}^{\infty} 2^j$	[0,0,1,0]
$\frac{a}{b}$	[0,0,0,1]

Now that we have decided the label representation, we should also decide how to read the image and how to feed it to the machine learning model. As we have discussed earlier, we are using python as a programming language for implementing our solution, using numpy library with it for storing the data is a very standard approach. While reading data from a folder we can read the images in a grayscale mode. When we are reading data from a folder, we can read it like this.

```
image_path = glob.glob(dataset_folder + '/*.png')
im_array = np.array([np.reshape(np.array(Image.open(img).convert('L'), 'f'), (image_size)) for img in image_path])
y_from_file = pickle.load(open(lable_data_file, "rb"))
```

In this example, dataset_folder is the folder with all the data images, image_size is the size of a flattened image, lable_data_file is the pickle file with all the labels.

For training a model on this data we are using a convolutional neural network. Network configuration is as follow.

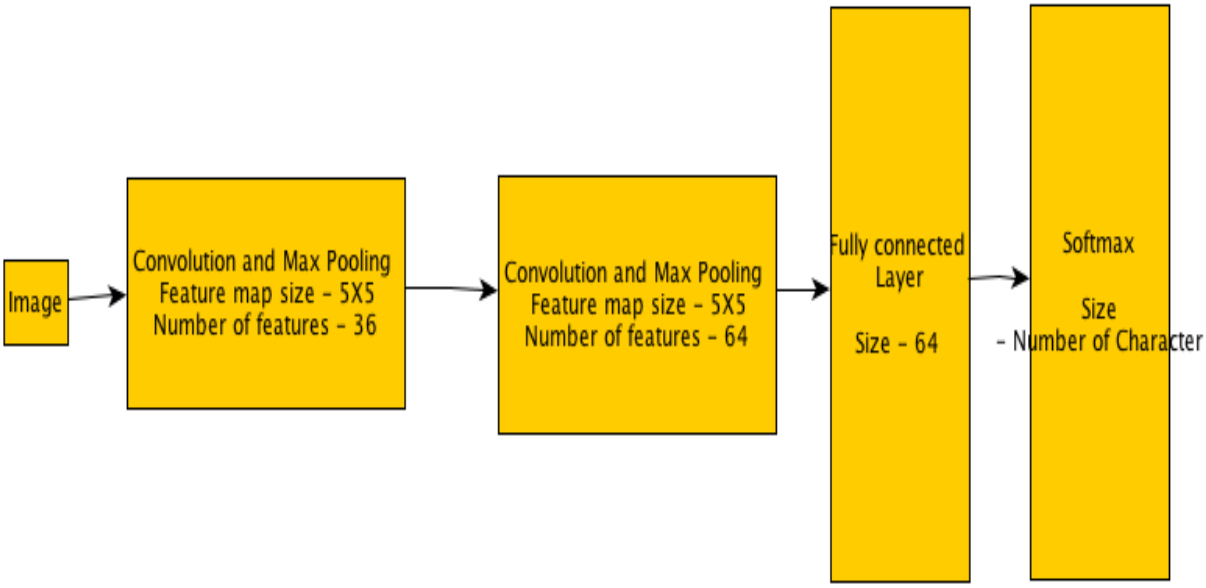

Figure 10: CNN architecture for prediction of the first character

While training this neural network we used Adam optimizer to update the weights in the neural network. There are multiple reasons for using Adam optimizer which we have covered in later parts of this report.

This implementation seemed perfectly fine in theory and we were expecting results to be good and model to converge a very small value of cost. In reality, this was not the case and to

our disappointment, model neither gives good results nor did it converge. There were few problems which we thought could be the reason for worry.

We compared our model with Yan LeCun's MNIST[8] model. Our model was trained on 20000 images compared to 70000 images LeCun's model was trained on. The size of an image in LeCun's model was 28 pixels in width and 28 pixels in height whereas mathematical equation images we generated were of size 60 pixels in width and 50 pixels in height. LeCun's MNIST data sample image has just one character in it whereas our data sample image had 5 to 10 characters. LeCun's model was trained on same neural network model transition as ours. From this data we concluded that we were not having enough data because of two primary reasons which are as follow:

1. Our image size was almost 4 times bigger than LeCun's MNIST image sizes.
2. Number of images in MNIST training set were 3.5 times more than the number of images in our training set

With this newfound reasoning, it seemed perfectly natural to have data set size of around 200,000 images to get good results and for the model to converge. So, our next experiment for the prediction of the first character was with 200,000 images of width 60 pixels and 50 pixels in height. As described previously, we had our own data generation setup. So, it was easy to generate as many images as we want. If this was not the case, then we would have struggled hard to find such images when required.

With the dataset of 200,000 images, we were extremely hopeful that model will converge and results would be as close as LeCun's results on MNIST dataset if not better than those. Even with this dataset, results were disappointing and the model did not converge. This experiment's disappointing results proved very good learning experience. This experiment's failure pushed us

to debug every possible step in the implementation. Even after rigorous debugging nothing worthy of change was not found. We studied multiple different implementations of image classification to identify any possible bugs in the code. Most of those implementations supported our approach and coding methodology of the implementations. After reading multiple papers it was realized that bug was neither there in the network configuration nor there in the TensorFlow code implementation. To substantiate our understanding, we conducted one more experiment.

In this experiment, we used the code we had written for mathematical equations' first character prediction with MNIST dataset which is incorporated in TensorFlow library. In this setting, model was converging within just 5000 images with an accuracy of more than 90%. This result was very important because that gave our model validation and us some validation that our implementation is quite accurate. This validation made us realize that model and network configuration is not the area of concern. So, it was natural to make sure we are reading data as it is being read by TensorFlow for MNIST images. There were two things which were of prime importance while reading data.

The first thing was the way we are assigning the value of 0 and 1 to pixels from an image. When we researched how data was being read by TensorFlow, it was seen that TensorFlow reads white pixels as value zero and black pixels as value 1. On the contrary, we were reading white pixel as value 1 and black pixel as value 0. The reason for this mistake was that we were using an inbuilt method from python PIL package to read the image in grayscale mode. We corrected this mistake by using little modification in the way we were reading the image data into numpy arrays. The following lines of code demonstrate how this modification in the code.

```
y = pickle.load(open(data_folder + ".p", "rb"))
images = []
```

```
y_return = []
for i in range(len(y)):
    image_path = data_folder + "/" + str(i) + ".png"
    current_image = np.reshape(np.asarray(Image.open(image_path).convert('1')),
image_size)
    current_image = current_image * -1.0 + 1.0
    images.append(current_image)
    y_return.append(y[i][character_number-1])
```

The second thing which we were doing while training was the way we were shuffling the data at the start of each epoch. Earlier we were shuffling the data manually with the help of a variable. So, there was a scope for overfitting because of that. We changed this approach by using an inbuilt method of another machine learning library sci-kit learn. Following lines of code demonstrate how shuffling can be done using sci-kit learn's shuffle method.

```
from sklearn.utils import shuffle
training_data, training_data_yi = shuffle(training_data, training_data_yi,
random_state=0)
```

After applying these two changes, we finally had our model to converge and accuracy of our model was more than 95%.

Once we achieved excellent results for the prediction of the first character in our experiment we decided to add more characters to see how well does this convolutional neural network classifies this big set of characters. For the final implementation of this deliverable we used the following sets of the characters.

1. Letters: It consisted of all the 26 lowercase alphabets of English language from ‘a’ to ‘z’.
2. Numbers: It consisted of all the 10 digits from ‘0’ to ‘9’.
3. Special symbols: It consisted fraction character (horizontal bar), summation symbol (∑), an integration symbol (∫), and a square root symbol (√).

With this new data of 20000 images, we trained the model using the convolutional neural network the same configuration as shown in the Figure 10 and we classified images into correct classes with an accuracy of more than 98% accuracy.

Predicting the Latex for Simple Equations

The first deliverable involving the prediction of the first character of a mathematical equation was a great milestone in the implementation of this project. After the stupendous results prediction of the first character, we decided to see if we could use the same network to predict all the characters in mathematical equations.

While generating the data for this deliverable it was important to limit the number of mathematical operations we were going to support. We decided to limit our scope to four operators namely, addition, subtraction, multiplication and, exponentials.

Network configuration for this case is as follow:

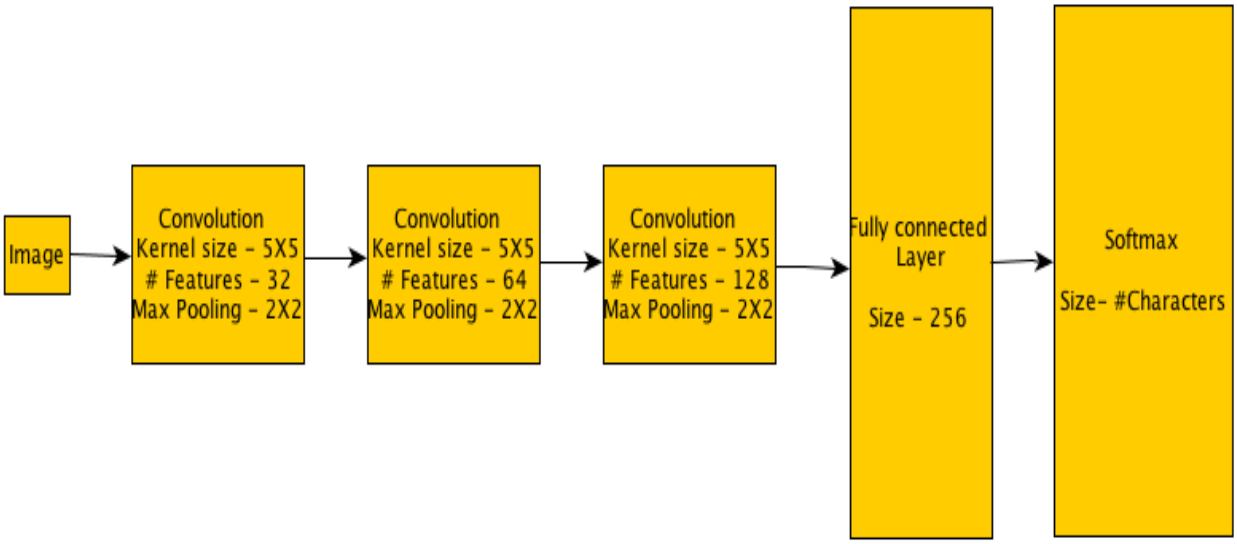

Figure 11: CNN architecture for the prediction of LaTeX for simple mathematical equations

As it can be seen from the figure 11, we have made this network deeper than the network we had earlier.

For this implementation, image size has got bigger because we were having images with seven characters. An important thing to understand in this scenario is that even though we were having 7 characters in the image, sometimes there were only 6 characters visible in the image. Why so? Consider an example: 2^3+5-a. In this example, there are 7 characters in the LaTeX representation but the image looks like this.

$$2^3 + 5 - a$$

Figure 12: Simple mathematical equation

The '^' symbol is not visible in the image. But, the convolutional neural network has to predict it. For this deliverable, we trained our model on 20000 images. In this case, we observed one pattern that whenever we were training it for characters which were deeper in the image it would take more time for the model to converge. We trained model for each character separately. For each character training, we made sure that model converged with the cost less than 0.5 and validation accuracy above 90%.

Predicting the LaTeX for Complex Mathematical Equations

The first two deliverables were the test to decide on whether our approach of using convolutional neural network was a correct approach to predict LaTeX expression for an image of the mathematical equation. For this deliverable, we were supposed to generate images which were more complex than the images we had generated earlier. To make mathematical equations complex we increased the number of characters in the mathematical expression. For this deliverable, we used around 50 characters. With these 50 characters as the possible characters we created mathematical equations containing maximum 30 character and generated images corresponding to those.

$$(4 - l) + 2^{18} * \sum_{w=5}^{\infty} w^2$$

Figure 13: A complex mathematical equation

As in this deliverable, we are using images which were having more than 50 characters, the number of filters in each layer has to be increased. So, we doubled the number of filters for each layer compared to the network configuration which we had for the previous deliverable. The following image shows the network configuration for this deliverable.

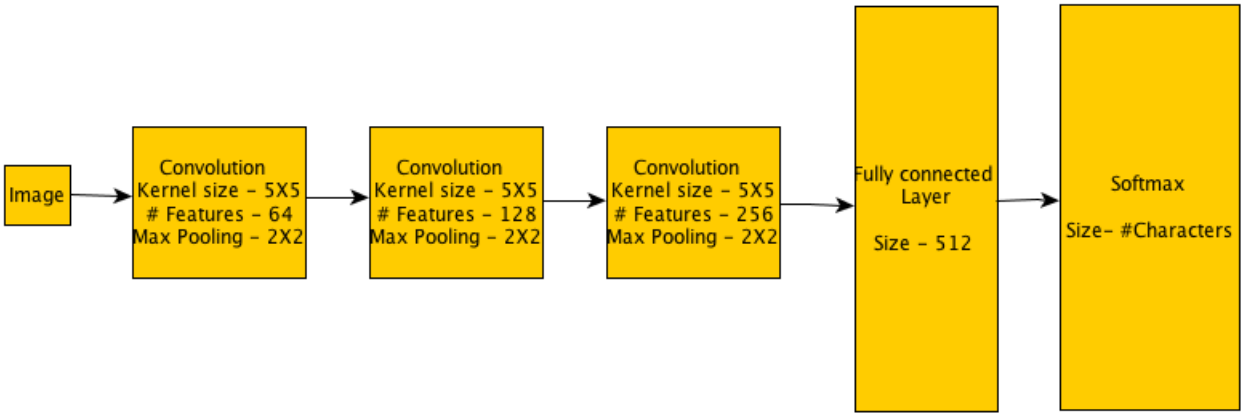

Figure 14: CNN architecture for the prediction of LaTeX for complex mathematical equation

With this network configuration, we trained our CNN model on 30000 images for each character separately. We trained it for 10 epochs. By the end of the training, we had achieved the accuracy of more 90% for each character and cost was reduced to 0.5. During the training of this network we learned how to avoid oscillations and how to choose different learning rate depending on the position of the character for which we were training the model. We have discussed more about it in the experiments sections.

Predicting the LaTeX for an Equation Containing Matrix Operations

After the successful experiments with complex mathematical equations we decided to explore more complex mathematical equations, which last for more than one line. One obvious choice for such equations was matrix equations. Matrix equations pose new challenges compared to other types of equations because of the arrangement as we discussed in the introduction section.

For our implementation purpose, we considered simple matrix operations involving matrix addition, subtraction multiplication, and assignment of a 2×2 matrix. The following image shows a visual representation of kind of mathematical equations for which we are training the model.

$$\begin{bmatrix} j & y \\ 3 & e \end{bmatrix} = \begin{bmatrix} 1 & i \\ z & v \end{bmatrix} * \begin{bmatrix} w & 9 \\ i & n \end{bmatrix}$$

Figure 15: Matrix operation

This matrix equation has 30 characters when we write it in the LaTeX format. We used the same network configuration which we used in the previous deliverable. In this implementation, as we had done earlier we trained the network for each character separately. In this deliverable too, we trained the network on 30000 images. We trained it for 10 epochs. By the end of the training we had achieved an accuracy of more 90% for each character and the cost was reduced to 0.5. During the training of this network, we learned how to avoid oscillations and how to choose different learning rate depending on the position of the character for which we were training the model. We have discussed more about it in the experiments sections.

EXPERIMENTS

Applied machine learning is an iterative process[9]. It is not so straightforward that we can get the desired results proposed model. Even for the most experienced machine learning scientists, it is very difficult to tell what exact values of the parameters would be the perfect values for generating the optimum results. This process follows multiple iterations where each iteration consists of the following steps as shown in the Figure 16. One has to go around the cycle many times to find a good choice for the network.

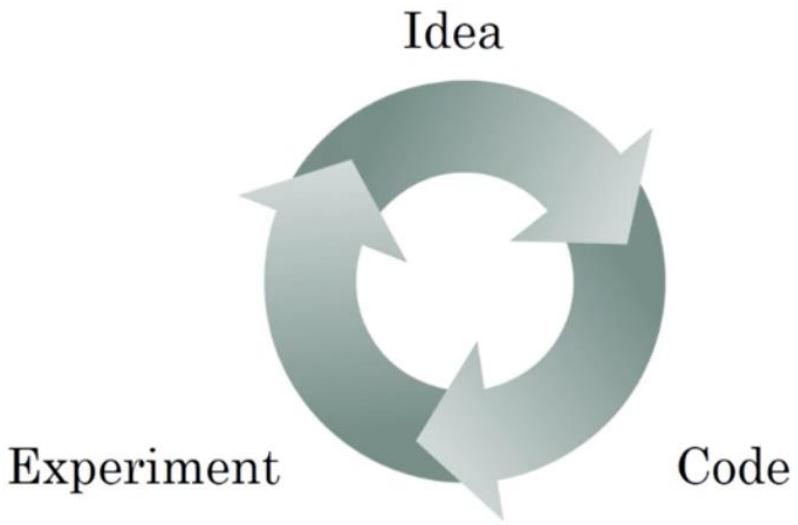

Figure 16: Iterative applied machine learning

In machine learning there are some variables whose value is set before the training of the model is started. This type of variables called as hyperparameters. On the contrary, parameters are the variables whose value is computed during the training of the machine learning model. For our model, we experimented with various hyperparameters and learned multiple important lessons from each of the experiments which we will summarize in the next few pages.

We will be specifically discussing the experiments we did with following hyperparameters.

1. Mini batch size
2. Size of feature map
3. Number of feature maps
4. Changing the value of learning rate

Mini Batch Size

There are two main reasons for deep learning to be successful in today's time. The first reason is, of course, the availability of powerful hardware. The second reason is the availability of the large datasets. Training a machine learning model on this big dataset is a slow process. Therefore, finding a fast way to train the machine learning model is very important in minimizing the time required to perform experiments and finding the most efficient configuration of the mode.

When we are using python as a programming language and numpy for vectorizing the input and the output, some part of the speed up process has been solved. There is still a lot of scope for speeding up. If we are using the entire training set consisting of thousands or hundreds of thousands of data points, then we have to process all the data points before deciding to move in the direction of the minimum value of the cost function using optimization algorithm. Similarly, for the next iteration, we have to process entire training set before making the decision about the next step in the direction of finding the minima.

There can be a better way to deal with this situation of processing the entire training dataset. Consider another approach where we can make the decision of movement in the direction of minimum cost even before processing the entire dataset. This can be achieved by splitting the training set into multiple smaller training sets. These smaller training sets are called as mini-batches. Consider an example where we have 50000 data points in the training set. We can divide this training set into 500 mini-batches where each mini-batch would have 100 data points. In this case, we randomly decided the mini-batch size to be of 100 data points. There can be different ways to decide the size of the mini-batch.

Deciding the size of the mini-batch is not so straightforward decision. There are a lot of different options available to select the size of the mini-batch. These options vary from having a mini-batch of single data point which is also called stochastic processing and the other extreme is using all the data points in the training data which is called as batch processing. Both the approaches have their advantages and disadvantages. Let us look at those before deciding the most optimum size of the mini-batch.

Table 2: Different Batch Size Advantages and Disadvantages

	Stochastic Processing	Batch processing	Mini-Batch processing
What it is?	Mini-batch size is one. It contains only one data point from the training set at a time. The number of mini-batches is equal to the number of examples in the training set	Mini-batch size equal to the data size of the training set. It contains all the data point from the training dataset. The Number of mini-batches is equal to one.	Mini-batch size is chosen based on experiments. Number of mini-batches = Training data size / min-batch size
Advantages	We can start seeing progress in the direction of the minimum cost from	We can move in direction of minimum cost with a very few	We utilize the speedup achieved through vectorization. We can start seeing the progress

	the first example itself.	oscillations. It is expected for the cost to go down with every single iteration.	in the movement towards the minimum cost value from the first batch itself
Disadvantages	We lose the speedup which we had achieved using vectorization.	It takes more time for training because we are taking one step in the direction of minima after processing all the examples.	There would be oscillations while moving towards the direction of minimum cost value.

As seen from the above table, we can see that dividing the data into mini-batches is an efficient way to train machine learning model. Typical mini-batch sizes used in the real-world machine learning training experiments are of sizes 64, 128, 256, 512 or 1024 data samples. We can see that all of these have one thing in common; they are powers of 2. As the computer memory is laid out and accessed in the memory blocks of the power of two, sometimes having the batch size power of 2 gives provides faster training [10].

Feature Map

In CNN, we have a sliding window which we slide over the image to identify different features of the image. This sliding window is called as feature map or kernel or feature detector. Usually, the feature map is represented using a multidimensional matrix. In case of a two-dimensional image it is a two-dimensional matrix of certain size. Depending on different values of the elements in this two-dimensional matrix we get different output of convolution when the image is convoluted with the feature map. The following images shows how the application of different filters will create different results.

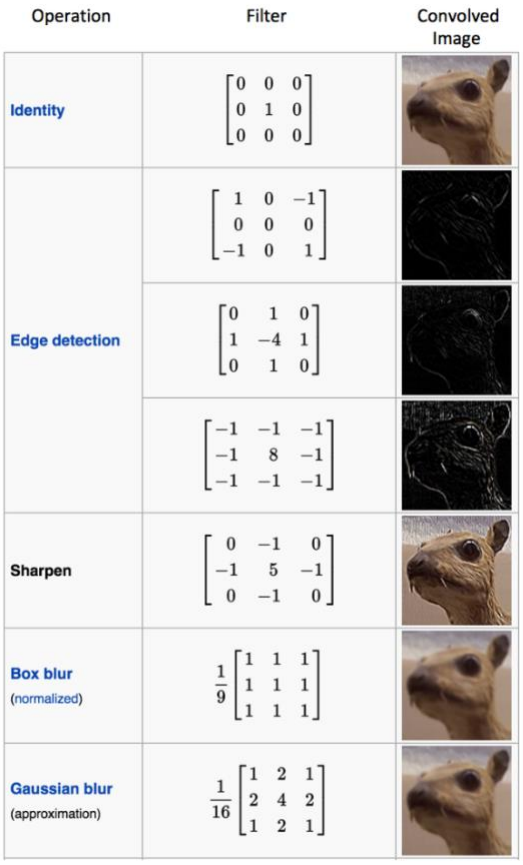

Figure 17: Filter application on an image

As shown in the example, we get a different convoluted image for different filters. These filter values are initialized to certain values before training. While training the machine learning model, these values modified so that they will identify the desired features.

Choice of feature map is an important decision because it is directly connected with the amount of time required to train the network and the kind of features to be recognized. As convolution operation is a dot product of feature map and a receptive field of an image, bigger the size of feature map more time it would take for training the model because there would those many numbers of calculations. Similarly, if we are trying to get the smaller features in an image having a smaller size of feature map helps.

As neural network training is an expensive operation, most of the times, it is recommended to take advantage of earlier research and apply transfer learning while deciding the hyperparameters of the neural network. Our problem is similar to the classification of digits with MNIST dataset, therefore it seemed obvious to use the kernel size as LeCun's CNN had used. We experimented with 3 sizes of feature map before settling onto the feature map size of 5X5 as used by LeCun in training MNIST data.

After deciding the size of feature map, it was important to decide the number of feature maps at each layer of CNN. As choosing the size of feature map based LeCun's CNN training of MNIST data proved to be useful, it seemed like a good option to use a comparative number of features. For example, LeNet was trying to classify just the digits, comparatively, our network is trying to classify more than 50 characters.

The following are the characters which we tried to classify.

'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+', '-', '*', '^', '\\int', '\\sum', '(', ')', '{', '}', '_', ' ', '=', '\\infty', '\\begin{bmatrix}', '\\end{bmatrix}', '\\\\', '&'

As we are having almost 50 characters to be classified the network layers we used were having more number of filters.

Learning Rate

As discussed earlier, we have used Adam optimizer [11] to update network weights because it is most popular for computer vision problems. While using any optimizer with TensorFlow, we have to provide a hyper parameter learning rate. This hyperparameter, decides the how network weights would be updated. This hyperparameter decides the magnitude of the steps taken towards the lowest cost. In theory, higher the value of learning rate faster we should reach the minimum cost. In reality, this is not the case because if we use a very high value of learning rate it might happen that our step towards lowest cost is so big that we would actually miss the minimum cost and move away from it. With the same learning rate, it might happen that we would miss it again and move away from it. This kind of behavior is also called as oscillating around the minimum cost. To avoid such behavior, it is important to decide on such value of learning rate that we each the local minimum cost without oscillating around it for too long.

In our implementation, while training the network for the prediction of the first 3 to 4 characters we kept the learning rate to be 0.01 and we achieved phenomenal results. We tried the same learning rate for the prediction of rest of the characters too expecting similar results. To our surprise, the model did not converge for the prediction of characters which were little away from the first few characters. The model kept oscillating for different value of cost for long period of time. This prompted us to choose smaller value of learning rate for the prediction of characters which were way from the first few characters. We used a various different value of learning rate depending on the position of the character in the equation.

Results

All the results are for data involving complex mathematical equations shown in Figure 13.

Mini batch size

For our convolutional neural network training, we experimented with a batch size of 64, 128, 512 and 1024 data samples. We got the most optimum results for the batch size of 128 data samples.

Feature maps size

The following table shows our observations about use of three different feature map sizes with which we experimented.

Table 3: Feature Map Size and Observation

Feature map size	Observation
5X5	It took around 10 seconds to train one batch of 128 images. Model converged and performed well on the test and validation data.
7X7	It took around 15 seconds to train on one batch of 128 images. Model worked fairly well on test and validation data but the efficiency was not as good as it was when we were using 5X5 filter size.
10X10	It took around 25 second to trains on one batch of 128 images. Even after training for more than 10 hours the model did not perform well on test data.

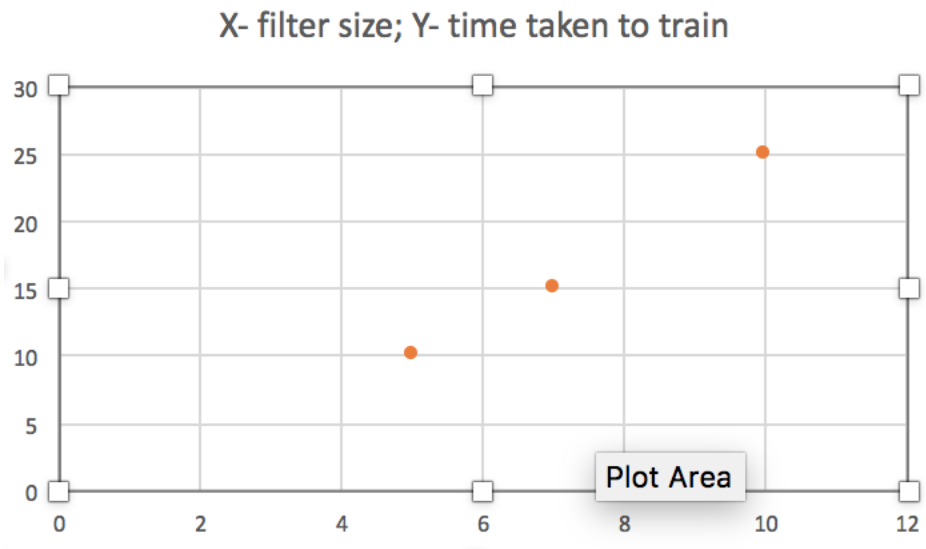

Figure 18: Feature map size vs time taken(sec)

Learning rate

The following table shows the various values of learning rate which were used for prediction of the character depending on the character position in the equation.

Table 4: Character Position and the learning rate

Character Numbers	Learning rate
1-4	0.01
5-10	0.0015
10-17	0.001
17-30	0.0001

Training cost: It takes more number of iteration to converge the model for twelfth character than the second character.

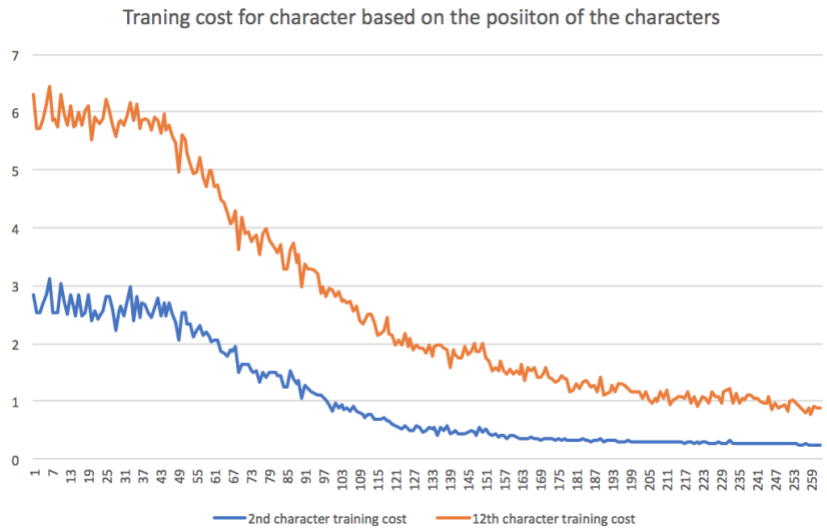

Figure 19: Training cost per iteration

Validation accuracy: The validation accuracy for model is better for the characters which are closer to the start than the characters which are deeper in the equation.

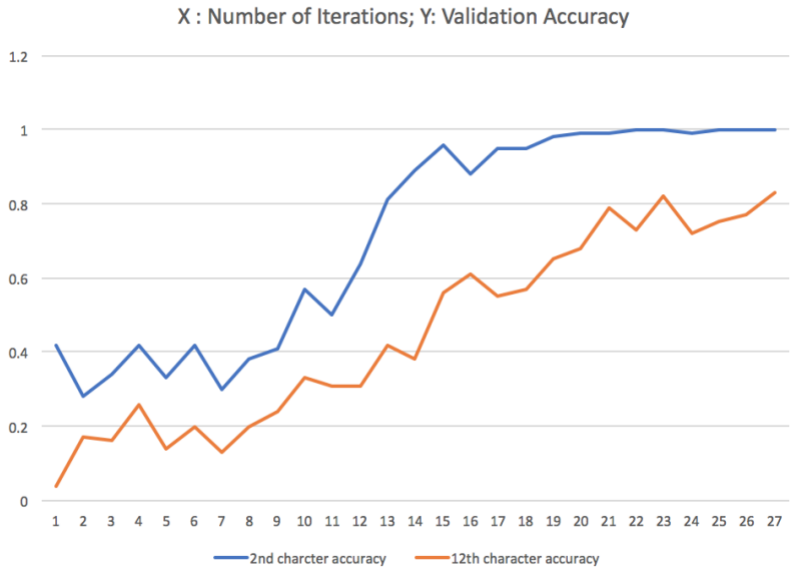

Figure 20: Validation accuracy over the epochs

CONCLUSION

In this project, we trained convolutional neural network to predict the LaTeX corresponding to an image of mathematical equations. We achieved an accuracy of more than 90% for the images containing up to 35 characters involving numbers, letters, and a few mathematical symbols. Our experiments involving the prediction of LaTeX corresponding to matrix operations of size 2×2 were also successful where we achieved an accuracy of more than 85%. In this project, we applied transfer learning from LeCun's CNN solution on MNIST dataset. This project was a very good exercise to see how far CNNs can go for the classification problems. While training the equations involving more than 35 characters we realized the limitations of using CNN for the classification of complex objects when there is a hard relationship between different objects in an image. Using an encoder–decoder model involving CNN for encoding and recurrent neural network for decoding would be a better approach for the prediction of LaTeX corresponding to an image. The encoder – decoder approach can be a future scope of this project.

REFERENCES

[1] "Emanuel Goldberg", Internet: https://history-computer.com/Internet/Dreamers/Goldberg.html

[2] Belaid, Abdelwaheb, and Jean-Paul Haton. "A syntactic approach for handwritten mathematical formula recognition." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 1 (1984): 105-111.

[3] Winkler, H-J., H. Fahrner, and Manfred Lang. "A soft-decision approach for structural analysis of handwritten mathematical expressions." *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*. Vol. 4. IEEE, 1995.

[4] Deng, Yuntian, et al. "Image-to-Markup Generation with Coarse-to-Fine Attention." *International Conference on Machine Learning*. 2017.

[5] Deng, Yuntian, Anssi Kanervisto, and Alexander M. Rush. "What You Get Is What You See: A Visual Markup Decompiler." *arXiv preprint arXiv:1609.04938* (2016).

[6] Sun, Chen, et al. "Revisiting unreasonable effectiveness of data in deep learning era." *arXiv preprint arXiv:1707.02968* 1 (2017).

[7] Goodfellow, Ian, et al. *Deep learning*. Vol. 1. Cambridge: MIT press, 2016.

[8] LeCun, Yann, et al. "Learning algorithms for classification: A comparison on handwritten digit recognition." *Neural networks: the statistical mechanics perspective* 261 (1995): 276.

[9] Le, Quoc V., et al. "On optimization methods for deep learning." *Proceedings of the 28th International Conference on International Conference on Machine Learning*. Omnipress, 2011.

[10] Ponnuru Rajeswari, Pookalangara Ajit Kumar “CIFAR-10 Classification using Intel® Optimization for TensorFlow*”, Internet: https://software.intel.com/en-us/articles/cifar-10-classification-using-intel-optimization-for-tensorflow, Dec 12, 2017

[11] Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." *arXiv preprint arXiv:1412.6980*(2014).